2015

ECE 254: Lab 5

RAMIE RAUFDEEN, SADMAN KHAN

INTER-THREAD COMMUNICATION VS INTER-PROCESS COMMUNICATION

Overview

This lab is an extension of lab four in addition to implementing another solution to the producer consumer problem. The first solution is to have threads communicate with each other using shared memory (global queue), in this case a queue. Just like the previous lab, this lab uses the POSIX API's functionalities; the second solution involved multiple processes utilizing the message queue from POSIX.

Timing Data

Inter-Thread Communication using Shared Memory

System Execution Time (X=500)

N	B	P	C	Time
100	4	1	1	0.000647
100	4	1	2	0.000746
100	4	1	3	0.000879
100	4	2	1	0.000681
100	4	3	1	0.00079
100	8	1	1	0.000624
100	8	1	2	0.00076
100	8	1	3	0.000834
100	8	2	1	0.000641
100	8	3	1	0.000869
398	8	1	1	0.001493
398	8	1	2	0.001809
398	8	1	3	0.002193
398	8	2	1	0.001672
398	8	3	1	0.002011

Standard Deviation of System Execution Time

N	B	P	C	Time
100	4	1	1	0.000266
100	4	1	2	0.00025
100	4	1	3	0.000212
100	4	2	1	0.0003
100	4	3	1	0.000297
100	8	1	1	0.000286
100	8	1	2	0.000266
100	8	1	3	0.000252
100	8	2	1	0.000291
100	8	3	1	0.000412
398	8	1	1	0.00041
398	8	1	2	0.00025
398	8	1	3	0.000365
398	8	2	1	0.000411
398	8	3	1	0.000508

Inter-Process Communication using POSIX Message Queue

System Execution Time (X=500)

N	B	P	C	Time
100	4	1	1	0.001663
100	4	1	2	0.001886
100	4	1	3	0.002291
100	4	2	1	0.001893
100	4	3	1	0.002173
100	8	1	1	0.001714
100	8	1	2	0.001953
100	8	1	3	0.00227
100	8	2	1	0.001863
100	8	3	1	0.002219
398	8	1	1	0.003084
398	8	1	2	0.003582
398	8	1	3	0.004163
398	8	2	1	0.003198
398	8	3	1	0.003477

Standard Deviation of System Execution Time

N	B	P	C	Time
100	4	1	1	0.00106
100	4	1	2	0.001185
100	4	1	3	0.001465
100	4	2	1	0.001269
100	4	3	1	0.001436
100	8	1	1	0.001144
100	8	1	2	0.001249
100	8	1	3	0.001492
100	8	2	1	0.001279
100	8	3	1	0.001419
398	8	1	1	0.001724
398	8	1	2	0.00183
398	8	1	3	0.002037
398	8	2	1	0.001605
398	8	3	1	0.001572

Case: (N, B, P, C) = (398, 8, 1, 3)

The chart below consists of the timing data for specific parameters.

	Inter-Thread	Inter-Process	Ratio
Average Execution Time (s)	0.00219318	0.00613259	2.796209
Standard Deviation (s)	0.000365435	0.000632096	N/A

From the ratio column above, one can conclude that inter-thread communication using shared memory is approximately 2.786209 times faster than inter-process communication with the POSIX message queue.

Outperformance Ratio

For all parameter values, inter-thread communication outperforms inter-process communication. The performance ratio column was calculated by dividing the execution time of inter-process communication by the execution time of inter-thread communication with the same parameters.

N	B	P	C	Performance Ratio
100	4	1	1	2.570324575
100	4	1	2	2.528150134
100	4	1	3	2.606370876
100	4	2	1	2.779735683
100	4	3	1	2.750632911
100	8	1	1	2.746794872
100	8	1	2	2.569736842
100	8	1	3	2.721822542
100	8	2	1	2.906396256
100	8	3	1	2.553509781
398	8	1	1	2.065639652
398	8	1	2	1.980099502
398	8	1	3	1.898312813
398	8	2	1	1.912679426
398	8	3	1	1.728990552

Averaging the performance ratio column yields approximately 2.42.

Please refer to the Excel workbook (lab5_rpt.xlsx) related to this lab for further information on calculations.

Discussion

From analyzing the timing data presented above, the average execution time in seconds for inter-thread communication using shared memory is approximately 2.8 times faster than inter-process communication using the POSIX message queue [for the requested case of (N, B, P, C) = (398, 8, 1, 3)] . Graphs were not presented for various parameter values because the inter-thread communication consistently outperformed the inter-process communication regardless of parameter change. From a theoretical perspective of the problem, it is important to consider that it is much easier for the OS to context-switch between threads than processes; primarily because threads are much more lightweight.

Further Considerations

An analysis beyond the scope of this report might include changing the value of number of integers produced; by looking at the outperformance ratio table, it seems the outperformance of inter-thread communication decreases as the number of integers produced increases. In addition, the buffer size was unable to exceed 10; the maximum size for the POSIX message queue on the ECE-Linux machines is 10. This could have impacted the results of inter-process communication as well.

Advantages and Disadvantages

Why use Threads?

The advantage of utilizing a multi-threaded communication approach as opposed to multi-process communication approach is primarily because the OS can quickly switch between threads compared to processes. Threads being lighter weight compared to processes can help a lot with that!

Threads also take less time to start than processes, processes require a call to *fork()* and *exec()*.

Threads use a lot less resources than processes because a new process has to copy memory which is a performance hit. With the use of a shared memory, the data inside it would be shared between all threads in the executable. Finally, in this case we see that the size of a shared memory queue is much larger than the POSIX message queue on ECE-Linux, which has a size limit of 10.

If performance/speed is an important factor, it would be ideal to use inter-thread communication!

Why use Processes?

Inter-process communication should be used over inter-thread communication primarily because of the POSIX message queue feature. The queue ensures that there would be minimal concurrency bugs between the processes. Due to the fact that threads run in the same virtual memory and share a common resource (in our case a queue), this could lead to concurrency issues between the shared memory which must be handled. Problems such as race conditions and deadlocks are handled using synchronization techniques which may not be optimal and lead to future bugs. With utilizing the POSIX message queue, the synchronization is already handled and there would be no concern about deadlocks and such.

Another advantage is that processes are run in different executables, whereas threads in this case run in the same executable. As a result, a data fault in one thread may lead to affecting other threads. However, with processes running different executables, a faulty process will not affect other processes.

From a developer perspective, debugging programs that use inter-process communication would be a lot easier than debugging inter-thread communication based programs. There wouldn't be a need to deal with problems such as synchronization errors and data corruption.

If reliability is key, it would be ideal to use inter-process communication!

Conclusion

Inter-thread communication should be taken advantage of due to its fast performance; due to its context-switching, and resource usage. Inter-process communication could be leveraged if one is seeking reliability at the cost of performance.

Source Code Listing

Consumer.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <mqueue.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>
#include <errno.h>
#include <sys/time.h>
#include <time.h>
#include <string.h>
#include <math.h>


int main(int argc, char *argv[]) {

    /* processing inputs */
    int number_of_messages = atoi(argv[1]);
    int cid = atoi(argv[2]);

    /* opening the message queue and the consumption queue */
    mqd_t qdes;
    char *qname = "/mailbox_lab4_extended";

    mqd_t consumption_queue;
    char *cmqname = "/consumption_queue_mailbox";

    qdes  = mq_open(qname, O_RDONLY);
    if (qdes == -1 ) {
        perror("mq_open()");
        exit(1);
    }

    consumption_queue = mq_open(cmqname, O_RDWR);
    if (consumption_queue == -1){
        perror("mq_open() failed from consumer");
        exit(1);
    }


    /* consume elements */

    int i;
    int count = 0;
    int sqrt_val;

    while(1){
        int recieved_message;
        int consumption_count;

        /* checking the number of messages that have been consumed in total.
           stopping the process if the consumption count == 0
        */

        mq_receive(consumption_queue, (char *)&consumption_count, sizeof(int), 0);
```

```
        if (consumption_count ==  0){
            mq_send(consumption_queue, (char *)&consumption_count, sizeof(int), 0);
            break;
        } else {
            consumption_count --;
            mq_send(consumption_queue, (char *)&consumption_count, sizeof(int), 0);
        }

        /* dequeue the message from the message queue */

        if (mq_receive(qdes, (char *)&recieved_message, sizeof(int), 0) == -1){
            printf("mq_receive() failed\n");
            return 1;
        }

        /* prints the values if perfect square */
        sqrt_val = sqrt(recieved_message);
        if((sqrt_val * sqrt_val) == recieved_message){
            printf("%i %i %i\n", cid, recieved_message, sqrt_val);
        }



    }

    /* closing the message queue on the consumer side */

    if (mq_close(qdes) == -1) {
        perror("mq_close() failed");
        exit(2);
    }

    /* closing the consumption queue on the consumer side */

    if(mq_close(consumption_queue) == -1){
        perror("mq_close() failed");
        exit(2);
    }

    return 0;

}
```

Producer.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <mqueue.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>
#include <errno.h>
#include <sys/time.h>
#include <time.h>
#include <string.h>


int main(int argc, char* argv[]){
    int pid;
    int total_message_number;
    int buffer_size;
    int producer_count;
    int consumer_count;


    total_message_number = atoi(argv[1]);
    pid = atoi(argv[2]);
    producer_count = atoi(argv[3]);
    consumer_count = atoi(argv[4]);

    /* Open the message queue from the producer side */

    mqd_t qdes;
    char *qname = "/mailbox_lab4_extended";

    qdes = mq_open(qname,  O_RDWR);
    if(qdes == -1){
        perror("mq_opne()");
        exit(1);
    }

    /* produce elements */

    int i;
    for(i = pid; i < total_message_number; i += producer_count){
        if(mq_send(qdes, (char*)&i, sizeof(int), 0) == -1){
            perror("mq_send() failed");
        }
    }

    /* close the message queue from the producer side */

    if (mq_close(qdes) == -1){
        perror("mq_close() failed");
        exit(2);
    }

    return 0;

}
```

Producer_Consumer_Process.c (Inter-Process method)

```c
#include <stdio.h>
#include <stdlib.h>
#include <mqueue.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>
#include <errno.h>
#include <sys/time.h>
#include <time.h>
#include <string.h>


double time_in_seconds(){
    struct timeval tv;
    double t1;

    gettimeofday(&tv, NULL);
    t1 = tv.tv_sec + tv.tv_usec/1000000.0;

    return t1;
}

/* spawn a child process
   Both the producers and consumers are each their own process
*/

int spawn (char* program, char** arg_list, int pid){
    arg_list[0] = program;


    /* need to cast the pid to a string to be passed into the process
       writing to arg_list[2] because the buffer count is not needed for the consumers or producers
    */
    char pid_string[10];
    sprintf(pid_string, "%d", pid);
    arg_list[2] = pid_string;


    pid_t child_pid;
    child_pid = fork();

    /* error check for child process */

    if (child_pid != 0){
        return child_pid;
    } else {
        execvp(program, arg_list);
        printf("%s\n", "an error has occurred in exec");
        abort();
    }
}

int main(int argc, char *argv []){

    /* process and ensure valid inputs */
    if (argc < 5){
```

```
58.         printf("%s\n", "Invalid number of inputs");
59.         exit(0);
60.     }
61.
62.     int consumer_count;
63.     int producer_count;
64.     int buffer_size;
65.     int total_message_number;
66.
67.     total_message_number = atoi(argv[1]);
68.     buffer_size = atoi(argv[2]);
69.     producer_count = atoi(argv[3]);
70.     consumer_count = atoi(argv[4]);
71.
72.     if (total_message_number < 0 || buffer_size < 0 || producer_count < 0 || consumer_c
    ount < 0){
73.         printf("%s\n", "All inputs must be positive integers" );
74.         exit(0);
75.     }
76.
77.     double starting_time;
78.     double finished_init_time;
79.     double ending_time;
80.     double execution_time;
81.
82.     /* create and open the message queue */
83.
84.     struct mq_attr attr;
85.     mqd_t qdes;
86.     mode_t mode = S_IRUSR | S_IWUSR;
87.
88.     attr.mq_maxmsg  = buffer_size;
89.     attr.mq_msgsize = sizeof(int);
90.     attr.mq_flags   = 0;
91.
92.     char *qname = "/mailbox_lab4_extended";
93.
94.     qdes = mq_open(qname, O_RDWR | O_CREAT, mode, &attr);
95.     if(qdes == -1){
96.         perror("mq_open() failed");
97.         exit(1);
98.     }
99.
100.          /* create and open a consumption queue
101.             consumption queue is a mq of size 1 which keeps track of the number of me
    ssages
102.             that have been consumed so far
103.          */
104.          struct mq_attr c_attr; //consumption queue attr
105.          mqd_t consumption_queue;
106.          mode_t mode_c = S_IRUSR | S_IWUSR;
107.
108.          c_attr.mq_maxmsg = 1;
109.          c_attr.mq_msgsize = sizeof(int);
110.          c_attr.mq_flags = 0;
111.
112.          char * consumption_queue_name = "/consumption_queue_mailbox";
113.
114.          consumption_queue = mq_open(consumption_queue_name, O_RDWR | O_CREAT, mode_c
    , &c_attr);
115.          if (consumption_queue == -1){
```

```
116.                perror("mq_open() for consumption failed");
117.                exit(1);
118.            }
119.
120.
121.            /* we start by writing the value of `total_message_number` into the consumpt
    ion_queue */
122.            if(mq_send(consumption_queue, (char *)&total_message_number, sizeof(int), 0)
     == -1){
123.                perror("mq_send() for consumption failed");
124.            }
125.
126.            starting_time = time_in_seconds();
127.
128.            /* loop through and create the producers and consumers processes */
129.
130.            int i;
131.
132.            for(i = 0; i < consumer_count; i++){
133.                spawn("./consume_p",argv, i);
134.            }
135.
136.            for(i = 0; i < producer_count; i++){
137.                spawn("./produce_p",argv, i);
138.            }
139.
140.
141.            finished_init_time = time_in_seconds();
142.
143.
144.            /* waiting for all the child processes to finsih before continuing
145.               reference: stack overflow
146.                */
147.            int pid;
148.            while (pid = waitpid(-1, NULL, 0)) {
149.               if (errno == ECHILD) {
150.                  break;
151.               }
152.            }
153.
154.            ending_time = time_in_seconds();
155.
156.            /* assumtion: execution time includes the time for initialization */
157.
158.            execution_time = ending_time - starting_time;
159.
160.            printf("System Excution time: %f seconds\n", execution_time);
161.
162.
163.            /* closing and unlinking the message queue and consumption queue */
164.
165.            if (mq_close(qdes) == -1){
166.                perror("mq_close() failed");
167.                exit(2);
168.            }
169.
170.            if(mq_close(consumption_queue) == -1){
171.                perror("mq_close() for cmq failed from main");
172.                exit(2);
173.            }
174.
```

```
175.            if (mq_unlink(qname) != 0) {
176.                perror("mq_unlink() failed");
177.                exit(3);
178.            }
179.
180.            if(mq_unlink(consumption_queue_name) == -1){
181.                perror("mq_unlink() failed for cmq from main");
182.                exit(3);
183.            }
184.
185.            return 0;
186.        }
```

Producer_Consumer_Thread.c (Inter-Thread method)

```c
/*
    ECE 254- lab 5
    Note, The structure of this code is based on
    advancedlinuxprogramming.com/alp-folder/alp-ch04-threads.pdf
    listing 4.12
    (Recomended reading from the pre lab)

    Sadman Khan
    Ramie Raufdeen
*/


#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <semaphore.h>
#include <sys/time.h>
#include <stdlib.h>
#include <math.h>

/* a queue struct implemented with a circular array
   it gets allocated as global variable and becomes shared memory
   that can be accessed by all threads
*/


struct queue {
    int *array;
    int head;
    int tail;
};

struct queue message_queue;
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;

sem_t message_count;           //makes sure that there is a message in the queue before the consumer tries to consume
sem_t consumed_messages_count; //allow the consumer thread to exit once all the messages have been consumed
sem_t empty_space;             //makes sure that there is enough space in the queue to handle another enqueue

int producer_count;
int total_number_of_messages;
int message_queue_size;

/* a one time init for the message queue and semaphores */

void init_message_queue(){
    message_queue.array = (int*) malloc (sizeof(int) * message_queue_size);
    message_queue.tail = 0;
    message_queue.head = 0;

    sem_init(&message_count, 0,0);
    sem_init(&consumed_messages_count,0,total_number_of_messages);
    sem_init(&empty_space, 0, message_queue_size);
}
```

```
56.
57. double time_in_seconds(){
58.     struct timeval tv;
59.     double t1;
60.
61.     gettimeofday(&tv, NULL);
62.     t1 = tv.tv_sec + tv.tv_usec/1000000.0;
63.
64.     return t1;
65. }
66.
67. /* consumer thread function
68.    keep consuming from the message queue until the `consumed_messsages_count` hits 0
69. */
70.
71. void* dequeue_message(void* arg){
72.     int c_id = (int) arg;
73.     while(1){
74.
75.         if(sem_trywait(&consumed_messages_count)){
76.             break;
77.         }
78.
79.         //wait for a message to be in the queue
80.         sem_wait(&message_count);
81.
82.         // lock the message queue
83.         pthread_mutex_lock (&lock);
84.         int value, sqrt_val;
85.
86.             message_queue.head++;
87.             if (message_queue.head == message_queue_size){
88.             message_queue.head = 0;
89.             }
90.
91.         /* printing the value if it is a perfect square */
92.
93.         value = message_queue.array[message_queue.head];
94.         sqrt_val = sqrt(value);
95.
96.         if (value == (sqrt_val * sqrt_val)){
97.             printf("%d %d %d\n", c_id, value, sqrt_val);
98.         }
99.
100.                pthread_mutex_unlock(&lock);
101.
102.                //signal producers that there is one less message in the message queue
103.                sem_post(&empty_space);
104.
105.
106.            }
107.            return NULL;
108.        }
109.
110.        /* producer thread function
111.           produces using the given algorithm from the manual
112.
113.           each producer produces a set number of ints
114.        */
115.
116.        void* enqueue_message (void* arg){
```

```
117.            int p_id = (int) arg;
118.            int i;
119.            for(i = p_id; i < total_number_of_messages; i += producer_count){
120.
121.                // make sure that the number of ints in the message queue does not excee
    d the buffer size
122.                sem_wait(&empty_space);
123.
124.                pthread_mutex_lock(&lock);
125.
126.                message_queue.tail++;
127.                if (message_queue.tail == message_queue_size){
128.                message_queue.tail = 0;
129.                }
130.
131.                message_queue.array[message_queue.tail] = i;
132.                pthread_mutex_unlock(&lock);
133.
134.                // let consumers know that there is a message in the queue
135.                sem_post(&message_count);
136.
137.            }
138.            return NULL;
139.
140.        }
141.
142.
143.        int main(int argc, char *argv[0]){
144.            if (argc < 5){
145.                printf("Invalid number of inputs\n");
146.                exit(0);
147.            }
148.
149.            int consumer_count;
150.
151.            total_number_of_messages = atoi(argv[1]);
152.            message_queue_size = atoi(argv[2]);
153.            producer_count = atoi(argv[3]);
154.            consumer_count = atoi(argv[4]);
155.
156.            if (total_number_of_messages < 0 || message_queue_size < 0 || producer_count
     < 0 || consumer_count < 0){
157.                printf("all args must be positive integers");
158.                exit(1);
159.            }
160.
161.            init_message_queue();
162.
163.            double starting_time;
164.            double ending_time;
165.            double total_time;
166.
167.            /* create an array of producer and consumer thread ids */
168.
169.            pthread_t producer_thread_id[producer_count];
170.            pthread_t consumer_thread_id[consumer_count];
171.
172.            starting_time = time_in_seconds();
173.
174.            /* create the producer and consumer threads */
175.
```

```
176.            int i;
177.            for(i = 0; i < producer_count; i++){
178.                pthread_create(&(producer_thread_id[i]), NULL, &enqueue_message, (void*)
    (i));
179.            }
180.
181.            for(i=0; i < consumer_count; i++){
182.                pthread_create(&(consumer_thread_id[i]), NULL, &dequeue_message, (void*)
    (i));
183.            }
184.
185.
186.            /* joing the producer and consumer threads to avoid exiting the main thread
187.               before the other threads have been completed
188.            */
189.
190.            for(i=0; i < producer_count; i++){
191.                pthread_join(producer_thread_id[i], NULL);
192.            }
193.
194.            for(i=0; i < consumer_count; i++){
195.                pthread_join(consumer_thread_id[i], NULL);
196.            }
197.
198.            ending_time = time_in_seconds();
199.            total_time = ending_time - starting_time;
200.
201.            printf("System execution time: %f seconds\n", total_time);
202.
203.            /* destroying the semaphores */
204.
205.            sem_destroy(&message_count);
206.            sem_destroy(&consumed_messages_count);
207.            sem_destroy(&empty_space);
208.
209.            return 0;
210.        }
```